Miles

*Ravikumar Patel 26645208*

*Evgeny Chulak 29683968*

*Alain Fallara 21739638*

*Richard Grand’Maison 26145965*

*Pargol Postareh 26428126*

[Email address]

Poker Equity Tool (PET)  
  
  
Milestone 3  
  
SOEN 343

**GitHub Link to Project’s Source Code**

<https://github.com/ivarletap/poker-prime>

**Summary of Project**

This is a poker hand equity calculator. This calculator can be used by a user to calculate the hand equity for different types of poker, including 5 Card Draw, Limit and No-Limit Hold’Em and Omaha.

The poker calculator also functions as a fast PokerStars and Full Tilt hand parser. Past hand histories are analyzed to create various statistics on player performance, past opponents and tournament success.

The project is still in development and is maintained. It has 2 contributors, 146 commits and 1 branch. This project was originally uploaded to GitHub on January 22 2012, and it has been updated regularly. The last modifications to the project were made on September 19th 2015.

# 

# 

# 

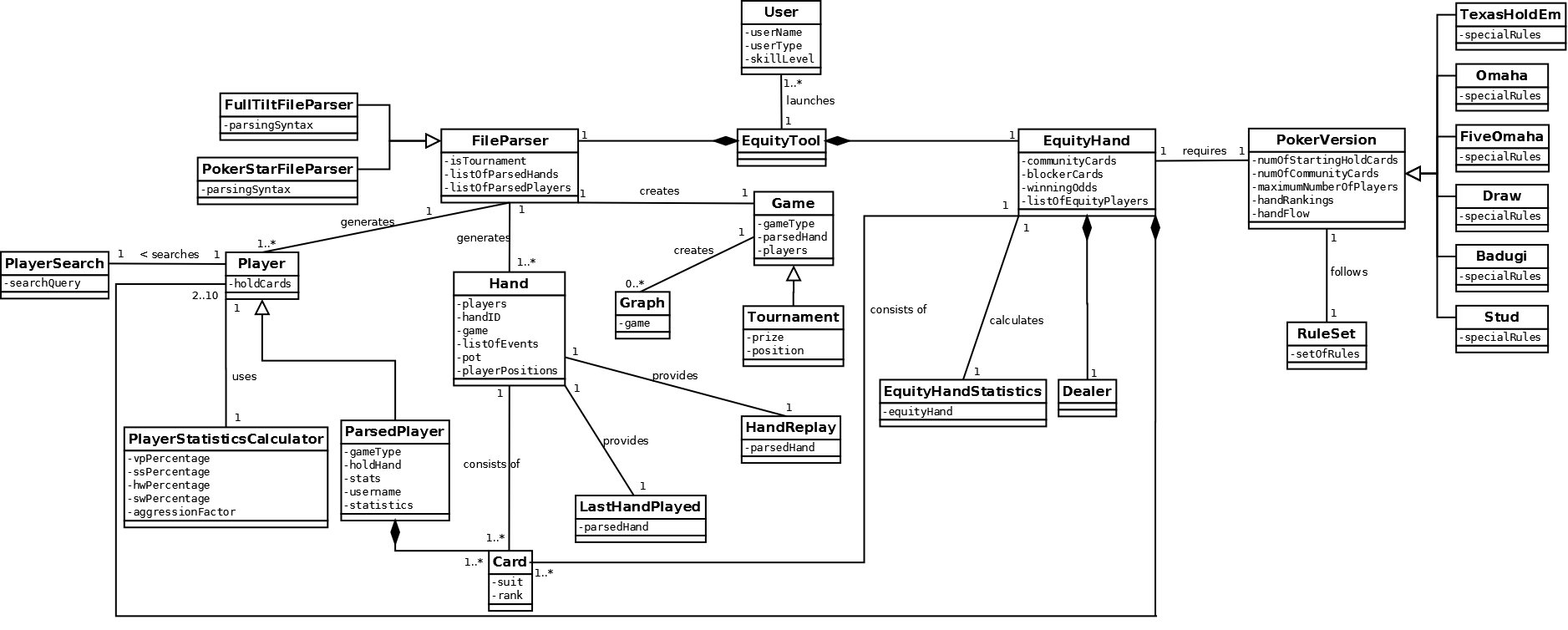
# 

# **Class Diagram of Actual System (for more clarity look at included png file)**

# 

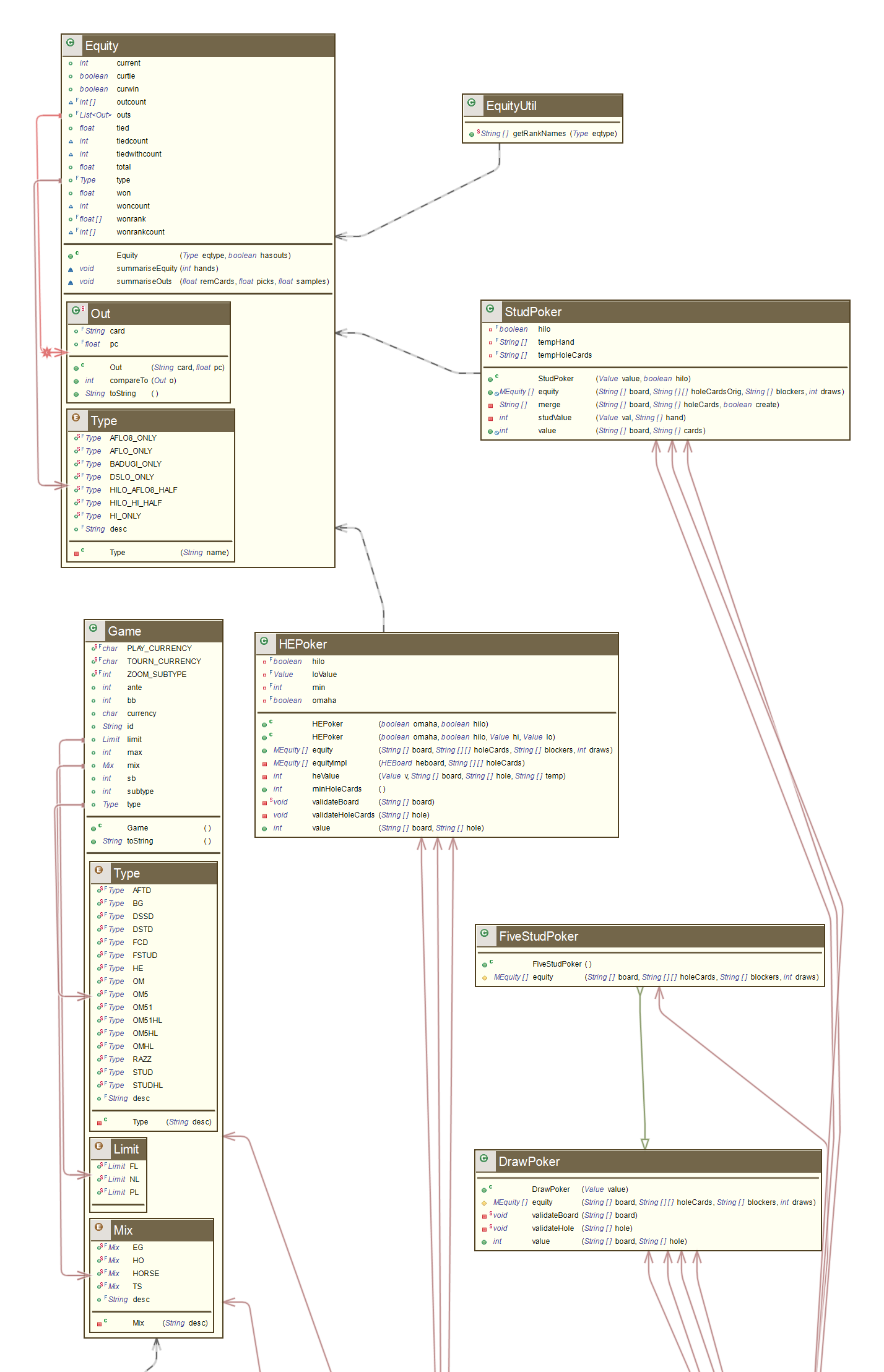
# **Figure 1- UML Diagram of Actual System Overview and Dependencies**

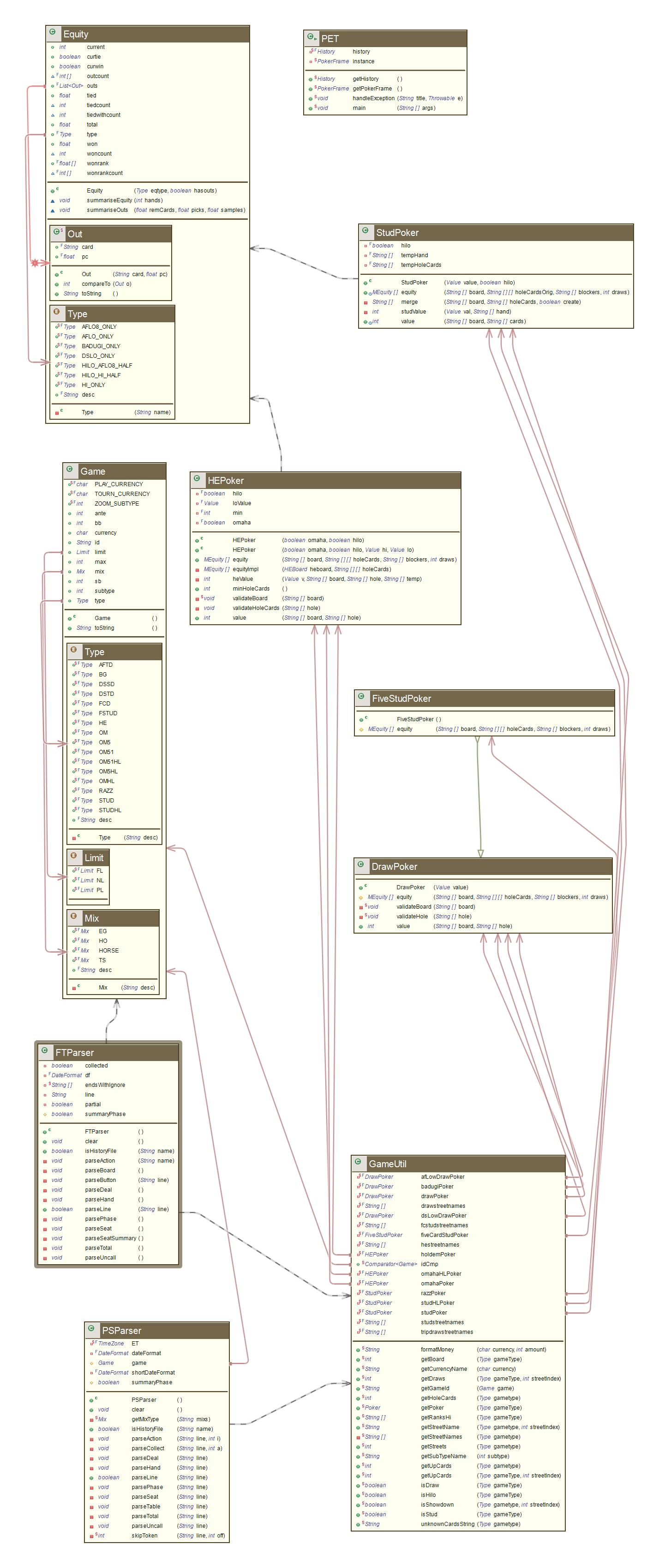
# **Conceptual UML Diagram**

  
 **Figure 2 - Conceptual Class Diagram (Milestone 2)**

**Extracted Class Diagram**

The extracted class diagram represents the classes we chose to study more closely.





**Figure 2 - Actual Class Diagram (Milestone 3)**

The class diagram was extracted from the Poker Equity Tool using AgileJ. The important classes were extracted in order to make a more readable class diagram, since the output of AgileJwas cluttered and much too intricate to be understood. The class diagram is centered on the Poker class and all relevant classes that are associated with it. For the complete architecture of the Poker Equity Tool, see the PNG file submitted along with this document.

**Class Description**

This section contains the description of the selected classes as well as their relationship to other classes.

**Poker Class**

The Poker class is the largest class of the Poker Equity Tool. It is responsible for constructing the conceptual representation of cards and poker hands. It is where the software determines the absolute and relative value of hands (i.e.: hand is a flush, and a flush beats a pair). This class is extended by many classes which must implement a few abstract methods related to equity calculations. The class isn't directly coupled to any other class despite it's low cohesion with the exception of being coupled to the String class. The reason is that decks, cards and poker types are represented by attributes of the class rather than separate objects.

**Game**

The Game class determines the type of poker of a particular game. A game can only have one type of Game. The game type is represented by an enum called Type. The enum Type serves as an attribute for the Hand class, and is used by the GameUtil class methods. This class's cohesion is high. It is used by several classes to determine the type of poker game.

**GameUtil**

This class has several methods which determine the value of many poker parameters based on which Game type is passed. It is coupled with the Game, DrawPoker, HEPoker, StudPoker, FiveStudPoker, Value and Poker classes. It is coupled to Game to determine the type of a specific poker game. With this information, GameUtil is coupled to DrawPoker, HEPoker, StudPoker, FiveStudPoker, Value and Poker by determining values based on a specific game.

**Equity**

This class creates objects representing the equity of a hand. It determines the type of equity to be measured (based on poker game type) and determines winning/tying percentages, available outs, and chances to win improve to win the hand. It is coupled to it's nested Out class which is able to compare cards based on percentages. This class's cohesion is high.

**FTParser and PSParser**

FTParser and PSParser respectively stand for FullTilt Parser and PokerStars parser. These two classes are designed to extract meaningful data from the automatically generated game logs of online poker games played on either the FullTilt or PokerStars platform. These classes are also subclasses of Parser2. These classes are also coupled to Action, DataFormat, Seat, Hand, Game, Matcher, TimeZone, GameUtil, Tourn and Date. Note that not all of the coupled classes were included in our extraction of the architecture because their importance being negligible. Thus coupling is high in both these classes.

**HEPoker, DrawPoker, FiveStudPoker and StudPoker**

These four classes subclasses (or children classes) of the Poker class. Their role is to provide the logic and mathematical formulae to measure various statistics based on which type of poker is played. Each game’s unique ruleset has an effect on probabilities and odds. All four of these classes are coupled to MEquity and MEquityUtil to calculate equity for a specific hand.

**Conceptual Classes compared to Actual Classes**

The conceptual class diagram differs from the actual class diagram mainly due to the architecture used in the Poker Equity Tool project. The conceptual class was not as detailed as the actual class diagram but it essentially contains all the important classes from the actual class. Generally speaking, the conceptual classes are each represented by several actual classes. Another reason why the conceptual classes don't match perfectly with the actual classes is that we vastly underestimated the complexity of mathematics involved in the various poker types.

The conceptual classes FullTiltParser, PokerStarsParser, ParsedPlayer, Hand and FileParser are present in the actual design, but they are split into a total of eight classes (FTHandRe, FTParser, HistoryUtil, Parser, Parser2, ParseUtil, PSHandRe and PSParser). This discrepancies arises because we underestimated how much information was contained in the logs and how different this information was parsed for various poker game types. In this case there are no real impacts on the system, but it simplifies the code by reducing method length and by separating concerns. For example, ParsedPlayer (from the conceptual class) is very similar to HistoryUtil (from the actual class).

The EquityHand class from the conceptual classes is represented by the Equity, EquityUtil, MEquity and MEquityUtil classes in the actual classes. As stated above, many of the conceptual classes have been broken down into multiple classes. We designing the conceptual architecture, we used the GUI provided by the Poker Equity Tool as a reference. There we decided to have a central PokerVersion class along with the types of poker games it supports (a total of 7 classes). However, in the actual architecture, we only have 6 classes (Game, GameUtil, DrawPoker, FiveStudPoker, HEPoker and StudPoker). The functionality of GameUtil was included in our PokerVersion class.

The Hand class of the conceptual design is represented by the Hand and HandUtil classes in the actual project. In our conceptual model, the Hand class is used both by the equity calculator and by the history segment of the program. The actual Hand and HandUtil classes are only used in the History part; hands are not explicitly described in the equity calculator.

**Relationship between 2 selected classes and sample source code**

The classes GameUtil and Game are related to each other in the following way. GameUtil references one Game. In other words, GameUtil is coupled to Game. The reason is the Game class determines the type of poker of a particular game. A game can only have one type of Game. The game type is represented by an enum called Type. The enum Type serves as an attribute for the Hand class, and is used by the GameUtil class methods. This class's coupling is high. It is used by several classes to determine the type of poker game.

**Class Gameutil:**

**public** **class** GameUtil {

//removed attributes for readability

**public** **static** String getGameId(Game game) {

StringBuilder sb = **new** StringBuilder();

sb.append(getCurrencyName(game.currency)).append(" ");

**if** (game.mix != **null**) {

sb.append(game.mix.desc).append(": ");

}

sb.append(game.type.desc).append(" ");

sb.append(game.limit).append(" ");

sb.append(game.max).append("-max "); // this is not the same as

**if** (game.subtype != 0) {

sb.append(getSubTypeName(game.subtype)).append(" ");

}

sb.append(formatMoney(game.currency, game.sb)).append("/");

sb.append(formatMoney(game.currency, game.bb));

**if** (game.ante != 0) {

sb.append("/");

sb.append(formatMoney(game.currency, game.ante));

}

**return** sb.toString();

}

**public** **static** **int** getHoleCards(Game.Type gametype) {

**switch** (gametype) {

//…

**default**:

**throw** **new** RuntimeException("unknown game type " + gametype);

}

}

**public** **static** **int** getUpCards(Game.Type gametype) {

**switch** (gametype) {

//…

**default**:

**throw** **new** RuntimeException();

}

}

**public** **static** String unknownCardsString(Game.Type gametype) {

**int** c = *getHoleCards*(gametype);

StringBuilder sb = **new** StringBuilder(c\*3);

**for** (**int** n = 0; n < c; n++) {

sb.append("[ ]");

}

**return** sb.toString();

}

**private** **static** String[] getStreetNames (Game.Type gametype) {

**switch** (gametype) {

//…

**default**:

**throw** **new** RuntimeException("no such game type " + gametype);

}

}

**public** **static** **boolean** isShowdown (Game.Type gametype, **int** streetIndex) {

**return** streetIndex == *getStreets*(gametype) - 1;

}

**public** **static** **int** getStreets (Game.Type gametype) {

**return** *getStreetNames*(gametype).length;

}

**public** **static** String getStreetName (Game.Type gametype, **int** streetIndex) {

**return** *getStreetNames*(gametype)[streetIndex];

}

**public** **static** Poker getPoker(Game.Type gameType) {

**switch** (gameType) {

//…

**default**:

**throw** **new** RuntimeException("no poker for game " + gameType);

}

}

**public** **static** String[] getRanksHi(Game.Type gameType) {

**switch** (gameType) {

//…

**default**:

**throw** **new** RuntimeException();

}

}

**public** **static** **int** getDraws(Game.Type gameType, **int** streetIndex) {

**switch** (gameType) {

//…

**default**:

**throw** **new** RuntimeException();

}

}

**public** **static** **int** getUpCards(Game.Type gameType, **int** streetIndex) {

**switch** (gameType) {

//…

**default**:

**throw** **new** RuntimeException();

}

}

**public** **static** **boolean** isDraw(Game.Type gameType) {

**return** *getDraws*(gameType, 0) > 0;

}

**public** **static** **boolean** isStud(Game.Type gameType) {

**switch** (gameType) {

//…

**default**:

**throw** **new** RuntimeException();

}

}

**public** **static** **boolean** isHilo(Game.Type gameType) {

**switch** (gameType) {

//…

**default**:

**throw** **new** RuntimeException();

}

}

**public** **static** **int** getBoard(Game.Type gameType) {

**switch** (gameType) {

//…

**default**:

**throw** **new** RuntimeException();

}

}

}

**Class Game:**

**public** **class** Game **implements** Serializable {

**public** **enum** Type {

***FCD***("5 Card Draw"),***HE***("Hold'em"),***OM***("Omaha"),***OMHL***("Omaha H/L"),***DSTD***("2-7 Triple Draw"),

***AFTD***("A-5 Triple Draw"), ***DSSD***("2Single Draw"), ***RAZZ***("Razz"), ***STUD***("7 Card Stud"),

***STUDHL***("7 Card Stud H/L"),***FSTUD***("5 Card Stud"),***OM51***("5+1 Card Omaha"),***OM5***("5 Card Omaha"),

***OM51HL***("5+1 Card Omaha H/L"),***OM5HL***("5 Card Omaha H/L"),***BG***("Badugi");

**public** **final** String desc;

**private** Type(String desc) {

**this**.desc = desc;

}

}

**public** **enum** Limit {

***NL***,***PL***,***FL***;

}

**public** **enum** Mix {

***HO***("Mixed HE/OM"),***TS***("Triple Stud"),***EG***("8-Game"),***HORSE***("HORSE");

**public** **final** String desc;

**private** Mix(String desc) {

**this**.desc = desc;

}

}

**public** **static** **final** **char** ***PLAY\_CURRENCY*** = 'p', ***TOURN\_CURRENCY*** = 't';

**public** **static** **final** **int** ***ZOOM\_SUBTYPE*** = 1;

**public** String id;

**public** Game.Mix mix;

**public** Game.Type type;

**public** **int** max;

**public** Game.Limit limit;

**public** **char** currency;

**public** **int** subtype;

**public** **int** sb, bb, ante;

**public** Game() {

}

@Override

**public** String toString () {

**return** "Game[" + id + "]";

}

}

# **Reverse Engineering Tools**

The UML based on the code-as-is was generated using a reverse engineering tool, AgileJ, which is a plugin for Eclipse. It is an easy plugin to use with many different configuration to produce different Class UML 2.0 Diagrams. Basically we need to create a new file, name it, and decide how the UML will be generated. There are 4 options: empty class diagram, populated by package, populate by inheritance, or populate by dependence. Also we need to decide to either include supertypes and subtypes, or outgoing and incoming dependencies. The classes of interest are selected and the tool uses the Java source code to automatically generate the Class Diagram. These diagrams will be updated following changes in the source code of the classes that are selected.

# **Code Smells and System Level Refactoring**

**Class: Poker.java**

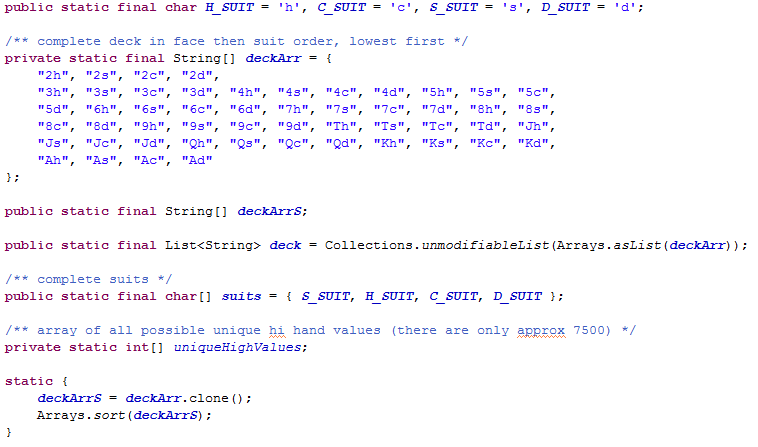
Code Smell: Long Class

Refactoring: Extract Class

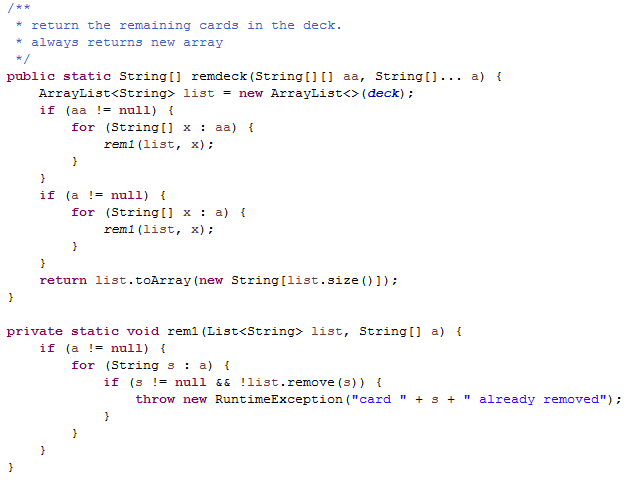
Explanation:

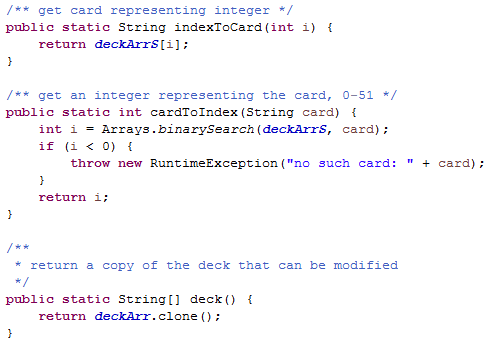
This class is used to determine a hand's value. To do so, it has several attributes that relate to a card deck entity. This entity doesn't have it's own class. To fix the code smell, we create a Deck class that contains all the attributes and methods related to a standard deck and the cards which it contains Next, we move the methods related to Deck and Card manipulation in the Deck class. The Poker class will then be only concerned about hand valuation, without knowing about Deck and Cards utility methods.

The attributes presented below are attributes of the Poker class which will be moved to the Deck class.

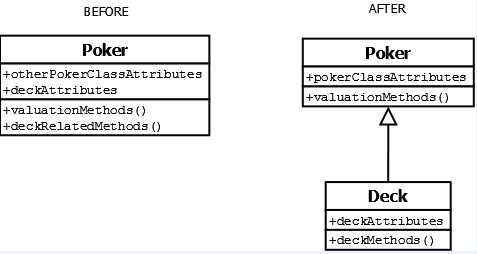


The following methods are those which will be moved from the Poker class to the Deck class.





The UML representation is presented below. The attributes and methods have been simplified for clarity. Naturally, the Poker class methods which use the new Deck object will have to be modified slightly to make the correct calls.

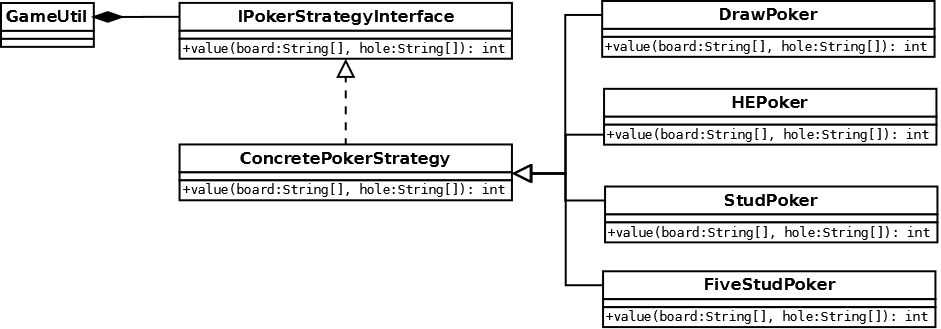


**Class: GameUtil.java**

Code Smell: Switch Statements (Type Checking)  
Refactoring Technique: Replace Type Code with Strategy Pattern  
Explanation:

GameUtil has 10 methods which have the code smell of switch statements. For this problem, we will focus on the getPoker() method of the class which returns a Poker instance without the use of polymorphism. To avoid this, we will discuss refactoring the getPoker() method using the Strategy pattern. Note that we have intentionally omitted many attributes and methods from the classes in the UML diagram below to simplify and clearly demonstrate our approach.

We designate the GameUtil class as the "context class" and create a "strategy" interface called IPokerStrategyInterface. This interface will be implemented by the ConcreteStrategyPoker class which will serve as parent class for DrawPoker, HEPoker, StudPoker and FiveStudPoker. This will eliminate having to instantiate and initialize each subclass of Poker individually. Polymorphism will determine (at runtime) the correct strategy required. This is the ideal architecture of this refactoring. However, it requires the creation, modification and deletion of many classes from the current architecture.



**Class: HEPoker.java**

Code Smell: Feature Envy

Refactoring: Move Method

Explanation:

Two methods, equityImpl() and heValue(), of this class handle data from object, HEBoard class, more than that of HEPoker, therefore they should be moved to HEBoard, in order to have related functions together. This will make the classes more internally coherent and will as a result diminished class dependence. This will also simplify the HEPoker class which is quite long.

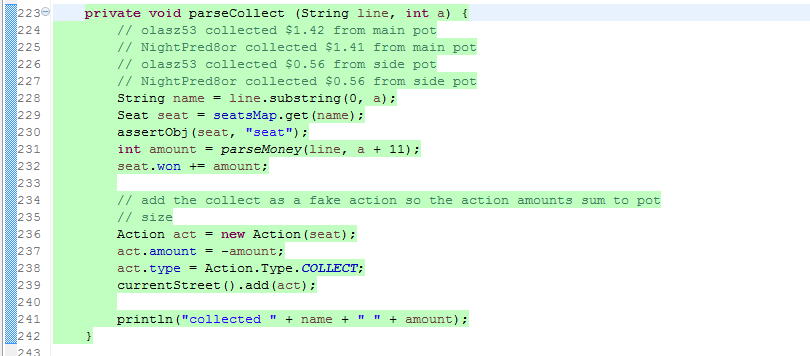
**PSParser Class**

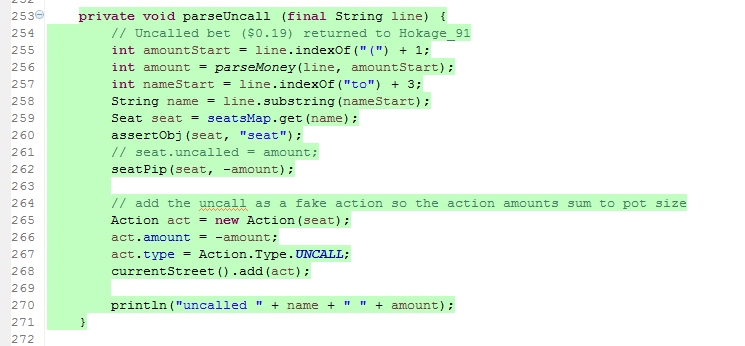
Code Smell: God Class

Refactoring: Extract Class

Explanation:

This is a class that has too many responsibilities. The affected methods that will be extracted are parseCollect() and parseUncall().





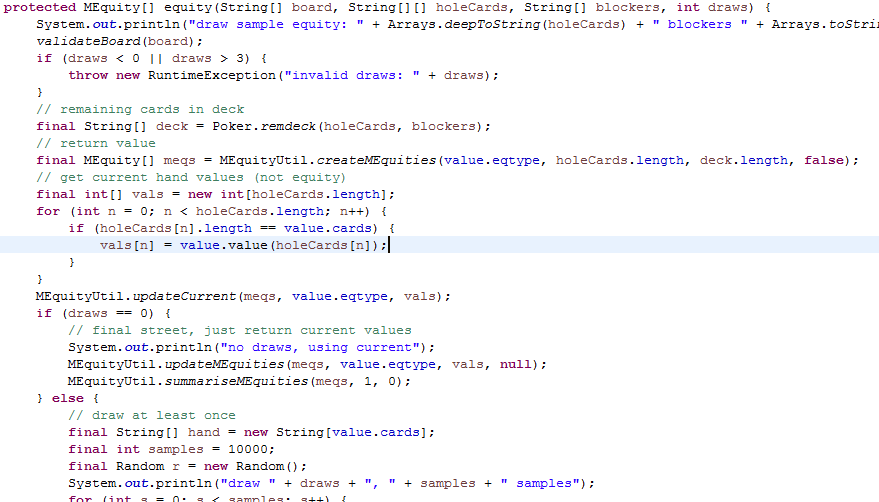
**DrawPoker Class**

Code Smell: Long Method

Refactoring: Extract Method

Explanation:

The equity method has over 60 physical lines of code and does several things. It is possible to extract methods to make it more readable. The original author left comments describing the various functionalities. These serve as good indicators to extract methods.



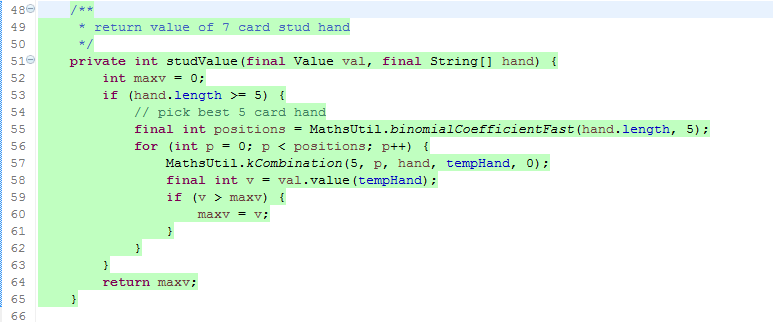
**StudPoker Class**

Code Smell: Feature Envy

Refactoring: Move Class

Explanation:

This method is used several time therefore it should be moved to the class Value in order to strengthen encapsulation. It could then be called from this class to get the necessary values.



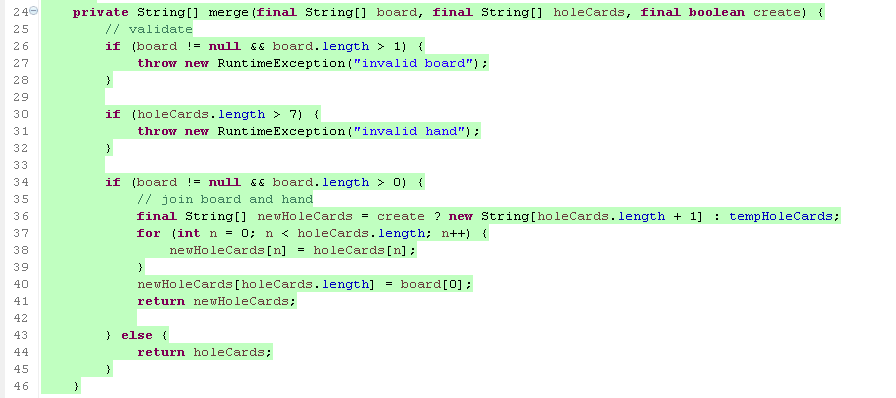
Code Smell: God Class

Refactoring: Extract Class

Explanation:

This class should only be used to define how calculations for Stud Poker equity are made. It should not be responsible for converting different hand types into other ones and, therefore this section of the code should have its own dedicated class. Therefore we will create a new class called StudPokerHandMerge to handle this operation and place this code inside. This will increase the cohesion of the StudPoker class.





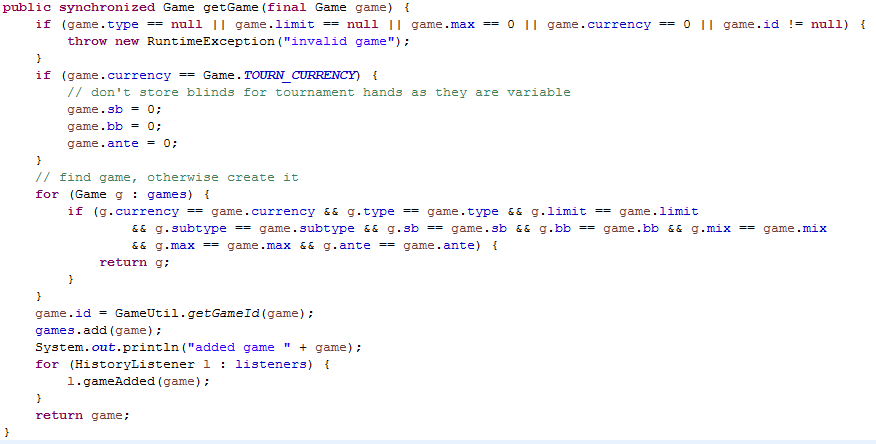
# **Specific Refactoring Implemented in Milestone 4**

**Class: History.java**

Code Smell: Long Method

Refactoring: Extract Method

Explanation:



The getGame(Game game) method tries to validate a game, determine if it is a tournament game, determine if it already exists in History, and if it isn't, it will add the game to History. It is possible.

To fix this code smell, we extract the first two if-statements into their own methods. We also extract the first for-statement into it's own method, but instead of returning the Game object if it is found, it returns a boolean. The last seven physical lines of code are also extracted into their method. This method will be called if the previous extracted method returns false; otherwise, Game g is returned.

**Class: GameUtil.java**

Code Smell: Switch Statements (Type Checking)  
Refactoring Technique: Replace Type Code with Strategy and Polymorphism  
Explanation:

Our ideal architecture mentioned in the previous section for this code smell requires extensive modification to the code. Again, we will focus on the getPoker() method of the class which returns a Poker instance without the use of polymorphism and a Strategy pattern. Thus we will refactor the getPoker() method of the GameUtil class using the Strategy pattern and Polymorphism. Since a drawback of the Strategy pattern is that we will have extra and unneeded classes, we propose to use existing classes from the project (see the next section). Note that we have intentionally omitted many attributes and methods from the classes in the both figures below to simplify and clearly demonstrate our approach. The figure below is the current implementation of the getPoker() method. As you can see, these is no use of Polymorphism in the switch statement and no use of the strategy pattern.



The UML diagram below represents our refactoring for this code smell. It involves designating the GameUtil class as the "context class" and designating the abstract Poker class will replace the "strategy interface" required for the Strategy pattern. This are using an abstract class instead of an interface due to convenience and not wanting to create unnecessary interface. The abstract method of Poker will be value() instead of execute() due not wanting to create an unnecessary method. The following classes will represent "concrete" strategies: DrawPoker, HEPoker, StudPoker and FiveStarPoker. This will eliminate having to instantiate and initialize each subclass of Poker individually. Polymorphism will determine (at runtime) the correct strategy required.

